Part SIX
Date/Time API

Chapter TWENTY-ONE
Core Date/Time Classes

Exam Objectives

Create and manage date-based and time-based events including a combination of date and time into a single object using LocalDate, LocalTime, LocalDateTime, Instant, Period, and Duration.
Define and create and manage date-based and time-based events using Instant, Period, Duration, and TemporalUnit.

A new Date/Time API

Since the beginning of Java, `java.util.Date` and `java.util.Calendar` (introduced later) have been the classes to use when working with dates and times.

However, these classes are far from perfect, some of their problems are:

- `java.util.Date` represents time with "only" milliseconds precision (which may not be enough in some applications). Years start from 1900 and months start at 0.
- The time zone of the date is the JVM's default time zone.
- Both `java.util.Date` and `java.util.Calendar` are mutable classes, meaning that when they change, they don't create another instance with the new values (which is not ideal now that you can program in a functional style with Java).

For those reasons, Java 8 introduced a new Date/Time API based on the popular date/time library Joda-Time and contained in the also new `java.time` package.

This chapter will deal with the core classes of the new API, that don't provide time zone information. The classes that do provide time zone information will be the topic of the next chapter.

Let's start with a high-level overview of the core classes.

All these classes are immutable, thread-safe, and with the exception of `Instant`, they don't store or represent a time-zone.

On the one hand, we have:

LocalDate
Represents a date with the year, month, and day of the month information. For example, *2015-08-25*.

LocalTime
Represents a time with hour, minutes, seconds, and nanoseconds information. For example, *13:21.05.123456789*.

LocalDateTime
A combination of the above. For example, *2015-08-25 13:21.05.12345*.

On the other hand:

Instant
Represents a single point in time in seconds and nanoseconds. For example *923,456,789 seconds and 186,054,812 nanoseconds*.

Period
Represents an amount of time in terms of years, months and days. For example, *5 years, 2 months and 9 days*.

Duration
Represents an amount of time in terms of seconds and nanoseconds. For example, *12.87656 seconds*.

`LocalDate`, `LocalTime`, `LocalDateTime` and `Instant` implement the interface `java.time.temporal.Temporal`, so they all have similar methods.

While `Period` and `Duration` implement the interface `java.time.temporal.TemporalAmount`, which also makes them very similar.

LocalDate Class

The key to learning how to use this class is to have in mind that it holds the year, month, day and derived information of a date. All of its methods use this information or have a version to work with each of them.

The following are the most important (most used) methods of this class.

To create an instance, we can use the static method of:

```
// With year(-999999999 to 999999999),
// month (1 to 12), day of the month (1 - 31)}
LocalDate newYear2001 = LocalDate.of(2001, 1, 1);
// This version uses the enum java.time.Month
LocalDate newYear2002 = LocalDate.of(2002, Month.JANUARY, 1);
```

Notice that unlike `java.util.Date`, months start from one. If you try to create a date with invalid values (like February, 29), an exception will be thrown. For today's date use `now()`:

```
LocalDate today = LocalDate.now();
```

Once we have an instance of `LocalDate`, we can get the year, the month, and the day with methods like the following:

```
int year = today.getYear();
int month = today.getMonthValue();
Month monthAsEnum = today.getMonth(); // as an enum
int dayYear = today.getDayOfYear();
int dayMonth = today.getDayOfMonth();
DayOfWeek dayWeekEnum = today.getDayOfWeek(); //as an enum
```

We can also use the get method:

```
int get(java.time.TemporalField field) // value as int
long getLong(java.time.TemporalField field) // value as long
```

Which takes an implementation of the interface `java.time.TemporalField` to access a specific field of a date. `java.time.ChronoField` is an enumeration that implements this interface, so we can have for example:

```
int year = today.get(ChronoField.YEAR);
int month = today.get(ChronoField.MONTH_OF_YEAR);
int dayYear = today.get(ChronoField.DAY_OF_YEAR);
int dayMonth = today.get(ChronoField.DAY_OF_MONTH);
int dayWeek = today.get(ChronoField.DAY_OF_WEEK);
long dayEpoch = today.getLong(ChronoField.EPOCH_DAY);
```

The supported values for `ChronoField` are:

- DAY_OF_WEEK
- ALIGNED_DAY_OF_WEEK_IN_MONTH
- ALIGNED_DAY_OF_WEEK_IN_YEAR
- DAY_OF_MONTH
- DAY_OF_YEAR
- EPOCH_DAY
- ALIGNED_WEEK_OF_MONTH
- ALIGNED_WEEK_OF_YEAR
- MONTH_OF_YEAR
- PROLEPTIC_MONTH YEAR_OF_ERA
- YEAR
- ERA

Using a different value will throw an exception. The same is true when getting a value that doesn't fit into an `int` with `get(TemporalField)`.

To check a `LocalDate` against another instance, we have three methods plus another one for leap years:

```
boolean after = newYear2001.isAfter(newYear2002); // false
boolean before = newYear2001.isBefore(newYear2002); // true
boolean equal = newYear2001.equals(newYear2002); // false
boolean leapYear = newYear2001.isLeapYear(); // false
```

Once an instance of this class is created, we cannot modify it, but we can create another instance from an existing one.

One way is through the `with()` method and its versions:

```
LocalDate newYear2003 = newYear2001.with(ChronoField.YEAR, 2003);
LocalDate newYear2004 = newYear2001.withYear(2004);
LocalDate december2001 = newYear2001.withMonth(12);
LocalDate february2001 = newYear2001.withDayOfYear(32);
```

```
// Since these methods return a new instance, we can chain them!
LocalDate xmas2001 = newYear2001.withMonth(12).withDayOfMonth(25);
```

Another way is by adding or subtracting guess what? Years, months, days, or even weeks:

```
// Adding
LocalDate newYear2005 = newYear2001.plusYears(4);
LocalDate march2001 = newYear2001.plusMonths(2);
LocalDate january15_2001 = newYear2001.plusDays(14);
LocalDate lastWeekJanuary2001 = newYear2001.plusWeeks(3);
LocalDate newYear2006 = newYear2001.plus(5, ChronoUnit.YEARS);

// Subtracting
LocalDate newYear2000 = newYear2001.minusYears(1);
LocalDate nov2000 = newYear2001.minusMonths(2);
LocalDate dec30_2000 = newYear2001.minusDays(2);
LocalDate lastWeekDec2001 = newYear2001.minusWeeks(1);
LocalDate newYear1999 = newYear2001.minus(2, ChronoUnit.YEARS);
```

Notice that the plus and minus versions take a `java.time.temporal.ChronoUnit` enumeration, different than `java.time.ChronoField` . The supported values are:

- DAYS
- WEEKS
- MONTHS
- YEARS
- DECADES
- CENTURIES
- MILLENNIA
- ERAS

Finally, the method `toString()` returns the date in the format *uuuu-MM-dd*:

```
System.out.println(newYear2001.toString()); // Prints 2001-01-01
```

LocalTime Class

The key to learning how to use this class is to have in mind that it holds the hour, minutes, seconds, and nanoseconds. All of its methods use this information or have a version to work with each of them.

The following are the most important (most used) methods of this class. As you can see, they are the same (or very similar) methods of `LocalDate` , adapted to work with time instead of date.

To create an instance, we can use the `static` method of:

```
// With hour (0-23) and minutes (0-59)
LocalTime fiveThirty = LocalTime.of(5, 30);
// With hour, minutes, and seconds (0-59)
LocalTime noon = LocalTime.of(12, 0, 0);
// With hour, minutes, seconds, and nanoseconds (0-999,999,999)
LocalTime almostMidnight = LocalTime.of(23, 59, 59, 999999);
```

If you try to create a time with an invalid value (like `LocalTime.of(24, 0)`), an exception will be thrown. To get the current time use `now()` :

```
LocalTime now = LocalTime.now();
```

Once we have an instance of `LocalTime`, we can get the hour, the minutes, and other information with methods like the following:

```
int hour = now.getHour();
int minute = now.getMinute();
int second = now. getSecond();
int nanosecond = now. getNano();
```

We can also use the `get()` method:

```
int get(java.time.TemporalField field) // value as int
long getLong(java.time.TemporalField field) // value as long
```

Just like in the case of `LocalDate`, we can have, for example:

```
int hourAMPM = now.get(ChronoField.HOUR_OF_AMPM); // 0 - 11
int hourDay = now.get(ChronoField.HOUR_OF_DAY); // 0 - 23
int minuteDay = now.get(ChronoField.MINUTE_OF_DAY); // 0 - 1,439
int minuteHour = now.get(ChronoField.MINUTE_OF_HOUR); // 0 - 59
int secondDay = now.get(ChronoField.SECOND_OF_DAY); // 0 - 86,399
int secondMinute = now.get(ChronoField.SECOND_OF_MINUTE);// 0 - 59
long nanoDay = now.getLong(ChronoField.NANO_OF_DAY);//0-86399999999
int nanoSecond = now.get(ChronoField.NANO_OF_SECOND);//0-999999999
```

The supported values for `ChronoField` are:

- NANO_OF_SECOND
- NANO_OF_DAY
- MICRO_OF_SECOND
- MICRO_OF_DAY
- MILLI_OF_SECOND
- MILLI_OF_DAY
- SECOND_OF_MINUTE
- SECOND_OF_DAY
- MINUTE_OF_HOUR
- MINUTE_OF_DAY
- HOUR_OF_AMPM
- CLOCK_HOUR_OF_AMPM
- HOUR_OF_DAY
- CLOCK_HOUR_OF_DAY
- AMPM_OF_DAY

Using a different value will throw an exception. The same is true when getting a value that doesn't fit into an `int` with `get(TemporalField)`.

To check a time object against another one, we have three methods:

```
boolean after = fiveThirty.isAfter(noon); // false
boolean before = fiveThirty.isBefore(noon); // true
boolean equal = noon.equals(almostMidnight); // false
```

Like `LocalDate`, once an instance of `LocalTime` is created we cannot modify it, but we can create another instance from an existing one.

One way is through the with method and its versions:

```
LocalTime ten = noon.with(ChronoField.HOUR_OF_DAY, 10);
LocalTime eight = noon.withHour(8);
LocalTime twelveThirty = noon.withMinute(30);
LocalTime thirtyTwoSeconds = noon.withSecond(32);
// Since these methods return a new instance, we can chain them!
LocalTime secondsNano = nooon.withSecond(20).withNano(999999);
```

Of course, another way is by adding or subtracting hours, minutes, seconds, or nanoseconds:

```
// Adding
LocalTime sixThirty = fiveThirty.plusHours(1);
LocalTime fiveForty = fiveThirty.plusMinutes(10);
LocalTime plusSeconds = fiveThirty.plusSeconds(14);
LocalTime plusNanos = fiveThirty.plusNanos(99999999);
LocalTime sevenThirty = fiveThirty.plus(2, ChronoUnit.HOURS);

// Subtracting
LocalTime fourThirty = fiveThirty.minusHours(1);
LocalTime fiveTen = fiveThirty.minusMinutes(20);
LocalTime minusSeconds = fiveThirty.minusSeconds(2);
LocalTime minusNanos = fiveThirty.minusNanos(1);
LocalTime fiveTwenty = fiveThirty.minus(10, ChronoUnit.MINUTES);
```

Notice that the plus and minus versions take a `java.time.temporal.ChronoUnit` enumeration, different than `java.time.ChronoField` . The supported values are:

- NANOS
- MICROS
- MILLIS
- SECONDS
- MINUTES
- HOURS
- HALF_DAYS

Finally, the method `toString()` returns the time in the format *HH:mm:ss.SSSSSSSSS*, omitting the parts with value zero (for example, just returning *HH:mm* if it has zero seconds/nanoseconds):

```
System.out.println(fiveThirty.toString()); // Prints 05:30
```

LocalDateTime Class

The key to learning how to use this class is to remember that it combines `LocalDate` and `LocalTime` classes.

It represents both a date and a time, with information like year, month, day, hours, minutes, seconds, and nanoseconds. Other fields, such as day of the year, day of the week, and week of year can also be accessed.

To create an instance, we can use either the static method `of()` or from a `LocalDate` or `LocalTime` instance:

```
// Setting seconds and nanoseconds to zero
LocalDateTime dt1 = LocalDateTime.of(2014, 9, 19, 14, 5);
// Setting nanoseconds to zero
LocalDateTime dt2 = LocalDateTime.of(2014, 9, 19, 14, 5, 20);
// Setting all fields
LocalDateTime dt3 = LocalDateTime.of(2014, 9, 19, 14, 5, 20, 9);
// Asumming this date LocalDate date = LocalDate.now();
```

```
// And this time LocaDate time = LocalTime.now();
// Combine the above date with the given time like this
LocalDateTime dt4 = date.atTime(14, 30, 59, 999999);
// Or this LocalDateTime dt5 = date.atTime(time);
// Combine this time with the given date. Notice that LocalTime
// only has this constructor to be combined with a LocalDate
LocalDateTime dt6 = time.atDate(date);
```

If you try to create an instance with an invalid value or date, an exception will be thrown. To get the current date/time use `now()` :

```
LocalDateTime now = LocalDateTime.now();
```

Once we have an instance of `LocalDateTime` , we can get the information with the methods we know from `LocalDate` and `LocalTime` , such as:

```
int year = now.getYear();
int dayYear = now.getDayOfYear();
int hour = now.getHour();
int minute = now.getMinute();
```

We can also use the `get()` method:

```
int get(java.time.TemporalField field)
long getLong(java.time.TemporalField field)
```

For example:

```
int month = now.get(ChronoField.MONTH_OF_YEAR);
int minuteHour = now.get(ChronoField.MINUTE_OF_HOUR);
```

The supported values for `ChronoField` are:

- NANO_OF_SECOND
- NANO_OF_DAY
- MICRO_OF_SECOND
- MICRO_OF_DAY
- MILLI_OF_SECOND
- MILLI_OF_DAY
- SECOND_OF_MINUTE
- SECOND_OF_DAY
- MINUTE_OF_HOUR
- MINUTE_OF_DAY
- HOUR_OF_AMPM
- CLOCK_HOUR_OF_AMPM
- HOUR_OF_DAY
- CLOCK_HOUR_OF_DAY
- AMPM_OF_DAY
- DAY_OF_WEEK
- ALIGNED_DAY_OF_WEEK_IN_MONTH
- ALIGNED_DAY_OF_WEEK_IN_YEAR
- DAY_OF_MONTH
- DAY_OF_YEAR
- EPOCH_DAY
- ALIGNED_WEEK_OF_MONTH
- ALIGNED_WEEK_OF_YEAR
- MONTH_OF_YEAR
- PROLEPTIC_MONTH

- YEAR_OF_ERA
- YEAR
- ERA

Using a different value will throw an exception. The same is true when getting a value that doesn't fit into an `int` with `get(TemporalField)`.

To check a `LocalDateTime` object against another one, we have three methods:

```
boolean after = now.isAfter(dt1); // true
boolean before = now.isBefore(dt1); // false
boolean equal = now.equals(dt1); // false
```

Once an instance of `LocalTime` is created, we cannot modify it, but we can create another instance from an existing one.

One way is through the with method and its versions:

```
LocalDateTime dt7 = now.with(ChronoField.HOUR_OF_DAY, 10);
LocalDateTime dt8 = now.withMonth(8);
// Since these methods return a new instance, we can chain them!
LocalDateTime dt9 = now.withYear(2013).withMinute(0);
```

Another way is by adding or subtracting years, months, days, weeks, hours, minutes, seconds, or nanoseconds:

```
// Adding
LocalDateTime dt10 = now.plusYears(4);
LocalDateTime dt11 = now.plusWeeks(3);
LocalDateTime dt12 = newYear2001.plus(2, ChronoUnit.HOURS);

// Subtracting
LocalDateTime dt13 = now.minusMonths(2);
LocalDateTime dt14 = now.minusNanos(1);
LocalDateTime dt15 = now.minus(10, ChronoUnit.SECONDS);
```

In this case, the supported values for `ChronoUnit` are:

- NANOS
- MICROS
- MILLIS
- SECONDS
- MINUTES
- HOURS
- HALF_DAYS
- DAYS
- WEEKS
- MONTHS
- YEARS
- DECADES
- CENTURIES
- MILLENNIA
- ERAS

Finally, the method `toString()` returns the date-time in the format *uuuu-MM-dd'T'HH:mm:ss.SSSSSSSSS*, omitting the parts with value zero, for example:

```
System.out.println(dt1.toString()); // Prints 2014-09-19T14:05
```

Instant Class

Although in practical terms, a `LocalDateTime` instance represents an instant in the time line, there is another class that may be more appropriate.

In Java 8, the `java.time.Instant` class represents an instant in the number of seconds that has passed since the epoch, a convention used in UNIX/POSIX systems and set at midnight of January 1, 1970 UTC time.

From that date, time is measured in 86,400 seconds per day. This information is stored as a `long`. The class also supports a nanosecond precision, stored as an `int`.

You can create an instance of this class with the following methods:

```
// Setting seconds
Instant fiveSecondsAfterEpoch = Instant.ofEpochSecond(5);
// Setting seconds and nanoseconds (can be negative)
Instant sixSecTwoNanBeforeEpoch = Instant.ofEpochSecond(-6, -2);
// Setting milliseconds after (can be before also) epoch
Instant fifthyMilliSecondsAfterEpoch = Instant.ofEpochMilli(50);
```

For the current instance of the system clock use:

```
Instant now = Instant.now();
```

Once we have an instance of `Instance`, we can get the information with the following methods:

```
long seconds = now.getEpochSecond(); // Gets the seconds
int nanos1 = now.getNano(); // Gets the nanoseconds
// Gets the value as an int
int milis = now.get(ChronoField.MILLI_OF_SECOND);
// Gets the value as a long
long nanos2 = now.getLong(ChronoField.NANO_OF_SECOND);
```

The supported `ChronoField` values are:

- `NANO_OF_SECOND`
- `MICRO_OF_SECOND`
- `MILLI_OF_SECOND`
- `INSTANT_SECONDS`

Using any other value will throw an exception. The same is true when getting a value that doesn't fit into an `int` with `get(TemporalField)`.

To check an `Instant` object against another one, we have three methods:

```
boolean after = now.isAfter(fiveSecondsAfterEpoch); // true
boolean before = now.isBefore(fiveSecondsAfterEpoch); // false
boolean equal = now.equals(fiveSecondsAfterEpoch); // false
```

Once an instance of this object is created, we cannot modify it, but we can create another instance from an existing one.

One way is through the with method:

```
Instant i1 = now.with(ChronoField.NANO_OF_SECOND, 10);
```

Another way is by adding or subtracting seconds, milliseconds, or nanoseconds:

```
// Adding
Instant dt10 = now.plusSeconds(400);
Instant dt11 = now.plusMillis(98622200);
Instant dt12 = now.plusNanos(3000138900);
Instant dt13 = newYear2001.plus(2, ChronoUnit.MINUTES);

// Subtracting
Instant dt14 = now.minusSeconds(2);
Instant dt15 = now.minusMillis(1);
Instant dt16 = now.minusNanos(1);
Instant dt17 = now.minus(10, ChronoUnit.SECONDS);
```

The supported `ChronoUnit` values are:

- NANOS
- MICROS
- MILLIS
- SECONDS
- MINUTES
- HOURS
- HALF_DAYS
- DAYS

Finally, the method `toString()` returns the instance in the format *uuuu-MM-dd'T'HH:mm:ss.SSSSSSSSS*, for example:

```
// Prints 1970-01-01T00:00:00.050Z
System.out.println(fifthyMilliSecondsAfterEpoch.toString());
```

Notice that it contains zone time information (Z). This is because `Instant` represents a point in time from the epoch of *1970-01-01Z* in the UTC zone time.

Period Class

The `java.time.Period` class represents an amount of time in terms of years, months and days.

You can create an instance of this class with the following of methods:

```
// Setting years, months, days (can be negative)
Period period5y4m3d = Period.of(5, 4, 3);
// Setting days (can be negative), years and months will be zero
Period period2d = Period.ofDays(2);
// Setting months (can be negative), years and days will be zero
Period period2m = Period.ofMonths(2);
// Setting weeks (can be negative). The resulting period will
// be in days (1 week = 7 days). Years and months will be zero
Period period14d = Period.ofWeeks(2);
// Setting years (can be negative), days and months will be zero
Period period2y = Period.ofDays(2);
```

A `Period` can be also thought as the difference between two `LocalDates`. Luckily, there's a method that support this concept:

```
LocalDate march2003 = LocalDate.of(2003, 3, 1);
LocalDate may2003 = LocalDate.of(2003, 5, 1);
Period dif = Period.between(march2003, may2003); // 2 months
```

The start date is **INCLUDED**, but **NOT** the end date.

Be careful about how the date is calculated.

First complete months are counted, and then the remaining number of days is calculated. The number of months is then split into years (1 years equals 12 months). A month is considered if the end day of the month is greater than or equal to the start day of the month.

The result of this method can be a negative period if the end is before the start (year, month and day will have a negative sign).

Here are some examples:

```
// dif1 will be 1 year 2 months 2 days
Period dif1 = Period.between( LocalDate.of(2000, 2, 10), LocalDate.of(2001, 4, 12));
// dif2 will be 25 days
Period dif2 = Period.between( LocalDate.of(2013, 5, 9), LocalDate.of(2013, 6, 3));
// dif3 will be -2 years -3 days
Period dif3 = Period.between( LocalDate.of(2014, 11, 3), LocalDate.of(2012, 10, 31));
```

Once we have an instance of `Period`, we can get the information with the following methods:

```
int days = period5y4m3d.getDays();
int months = period5y4m3d.getMonths();
int year = period5y4m3d.getYears();
int days2 = period5y4m3d.get(ChronoUnit.DAYS);
```

The supported `ChronoUnit` values are:

- DAYS
- MONTHS
- YEARS

Using any other value will throw an exception.

Once an instance of `Period` is created, we cannot modify it, but we can create another instance from an existing one.

One way is through the with method and its versions:

```
Period period8d = period2d.withDays(8);
// Since these methods return a new instance, we can chain them!
Period period2y1m2d = period2d.withYears(2).withMonths(1);
```

Another way is by adding or subtracting years, months, or days:

```
// Adding
Period period9y4m3d = period5y4m3d.plusYears(4);
Period period5y7m3d = period5y4m3d.plusMonths(3);
Period period5y4m6d = period5y4m3d.plusDays(3);
Period period7y4m3d = period5y4m3d.plus(period2y);

// Subtracting
Period period5y4m3d = period5y4m3d.minusYears(2);
Period period5y4m3d = period5y4m3d.minusMonths(1);
Period period5y4m3d = period5y4m3d.minusDays(1);
Period period5y4m3d = period5y4m3d.minus(period2y);
```

Methods plus and minus take an implementation of the interface `java.time.temporal.TemporalAmount` (i.e. another instance of `Period` or an instance of `Duration`).

Finally, the method `toString()` returns the period in the format `PNYNMND`, for example:

```
System.out.println(period5y4m3d.toString()); // Prints P5Y4M3D
```

A zero period will be represented as zero days, `P0D`.

Duration Class

The `java.time.Duration` class is like the `Period` class, the only thing is that it represents an amount of time in terms of seconds and nanoseconds.

You can create an instance of this class with the following of methods:

```
Duration oneDay = Duration.ofDays(1); // 1 day = 86400 seconds
Duration oneHour = Duration.ofHours(1); // 1 hour = 3600 seconds
Duration oneMin = Duration.ofMinutes(1); // 1 minute = 60 seconds
Duration tenSeconds = Duration.ofSeconds(10);
// Set seconds and nanoseconds (if they are outside the range
// 0 to 999,999,999, the seconds will be altered, like below)
Duration twoSeconds = Duration.ofSeconds(1, 1000000000);
// Seconds and nanoseconds are extracted from the passed milisecs
Duration oneSecondFromMilis = Duration.ofMilis(2);
// Seconds and nanoseconds are extracted from the passed nanos
Duration oneSecondFromNanos = Duration.ofNanos(1000000000);
Duration oneSecond = Duration.of(1, ChronoUnit.SECONDS);
```

Valid values of `ChronoUnit` are:

- NANOS
- MICROS
- MILLIS
- SECONDS
- MINUTES
- HOURS
- HALF_DAYS
- DAYS

A `Duration` can also be created as the difference between two implementations of the interface `java.time.temporal.Temporal`, as long as they support seconds (and for more accuracy, nanoseconds), like `LocalTime`, `LocalDateTime`, and `Instant`. So we can have something like this:

```
Duration dif = Duration.between( Instant.ofEpochSecond(123456789), Instant.ofEpochSecor
```

The result can be negative if the end is before the start. A negative duration is expressed with a negative sign in the seconds part. For example, a duration of -100 nanoseconds is stored as -1 second plus 999,999,900 nanoseconds.

If the objects are of different types, then the duration is calculated based on the type of the first object. But this only works if the first argument is a `LocalTime` and the second is a `LocalDateTime` (because it can be converted to `LocalTime`). Otherwise, an exception is thrown.

Once we have an instance of `Duration`, we can get the information with the following methods:

```
// Nanoseconds part the duration, from 0 to 999,999,999
int nanos = oneSecond.getNano();
// Seconds part of the duration, positive or negative
int seconds = oneSecond.getSeconds();
// Supports SECONDS and NANOS.Other units throw an exception
int oneSec = oneSecond.get(ChronoUnit.SECONDS);
```

Once an instance of `Duration` is created, we cannot modify it, but we can create another instance from an existing one.

One way is through the with method and its versions:

```
Duration duration1sec8nan = oneSecond.withNanos(8);
Duration duration2sec1nan = oneSecond.withSeconds(2).withNanos(1);
```

Another way is by adding or subtracting days, hours, minutes, seconds, milliseconds, or nanoseconds:

```
// Adding
Duration plus4Days = oneSecond.plusDays(4);
Duration plus3Hours = oneSecond.plusHours(3);
Duration plus3Minutes = oneSecond.plusMinutes(3);
Duration plus3Seconds = oneSecond.plusSeconds(3);
Duration plus3Millis = oneSecond.plusMillis(3);
Duration plus3Nanos = oneSecond.plusNanos(3);
Duration plusAnotherDuration = oneSecond.plus(twoSeconds);
Duration plusChronoUnits = oneSecond.plus(1, ChronoUnit.DAYS);

// Subtracting
Duration minus4Days = oneSecond.minusDays(4);
Duration minus3Hours = oneSecond.minusHours(3);
Duration minus3Minutes = oneSecond.minusMinutes(3);
Duration minus3Seconds = oneSecond.minusSeconds(3);
Duration minus3Millis = oneSecond.minusMillis(3);
Duration minus3Nanos = oneSecond.minusNanos(3);
Duration minusAnotherDuration = oneSecond.minus(twoSeconds);
Duration minusChronoUnits = oneSecond.pminus(1, ChronoUnit.DAYS);
```

Methods `plus` and `minus` take either another `Duration` or a valid `ChronoUnit` value (the same values used to create an instance).

Finally, the method `toString()` returns the duration with the format *PTnHnMnS*. Any fractional seconds are placed after a decimal point in the seconds section. If a section has a zero value, it's omitted. For example:

```
2 days 4 minutes PT48H4M
45 seconds 99 milliseconds PT45.099S
```

Key Points

- `LocalDate`, `LocalTime`, `LocalDateTime`, `Instant`, `Period`, `Duration` are the core classes of the new Java Date/Time API, located in the package `java.time`. They are immutable, thread-safe, and with the exception of `Instant`, they don't store or represent a time-zone.
- `LocalDate`, `LocalTime`, `LocalDateTime` and `Instant` implement interface `java.time.temporal.Temporal`, so they all have similar methods. While `Period` and

`Duration` implement interface `java.time.temporal.TemporalAmount`, which also makes them very similar.

- `LocalDate` represents a date with the year, month, and day of the month information. You can create an instance using:

```
LocalDate.of(2015, 8, 1);
```

- Valid `ChronoField` values to use with the `get()` method are: `DAY_OF_WEEK`, `ALIGNED_DAY_OF_WEEK_IN_MONTH`, `ALIGNED_DAY_OF_WEEK_IN_YEAR`, `DAY_OF_MONTH`, `DAY_OF_YEAR`, `EPOCH_DAY`, `ALIGNED_WEEK_OF_MONTH`, `ALIGNED_WEEK_OF_YEAR`, `MONTH_OF_YEAR`, `PROLEPTIC_MONTH`, `YEAR_OF_ERA`, `YEAR`, and `ERA`.
- Valid `ChronoUnits` values to use with the `plus()` and `minus()` methods are: `DAYS`, `WEEKS`, `MONTHS`, `YEARS`, `DECADES`, `CENTURIES`, `MILLENNIA`, and `ERAS`.
- `LocalTime` represents a time with hour, minutes, seconds, and nanoseconds information. You can create an instance using:

```
LocalTime.of(14, 20, 50, 99999);
```

- Valid `ChronoField` values to use with the `get()` method are: `NANO_OF_SECOND`, `NANO_OF_DAY`, `MICRO_OF_SECOND`, `MICRO_OF_DAY`, `MILLI_OF_SECOND`, `MILLI_OF_DAY`, `SECOND_OF_MINUTE`, `SECOND_OF_DAY`, `MINUTE_OF_HOUR`, `MINUTE_OF_DAY`, `HOUR_OF_AMPM`, `CLOCK_HOUR_OF_AMPM`, `HOUR_OF_DAY`, `CLOCK_HOUR_OF_DAY`, and `AMPM_OF_DAY`.
- Valid `ChronoUnits` values to use with the `plus()` and `minus()` methods are: `NANOS`, `MICROS`, `MILLIS`, `SECONDS`, `MINUTES`, `HOURS`, and `HALF_DAYS`.
- `LocalDateTime` is a combination of `LocalDate` or `LocalTime`. You can create an instance using:

```
LocalDateTime.of(2015, 8, 1, 14, 20, 50, 99999);
```

- Valid `ChronoField` and `ChronoUnits` values are a combination of the ones used for `LocalDate` and `LocalTime`.
- `Instant` represents a single point in time in seconds and nanoseconds. You can create an instance using:

```
Instant.ofEpochSecond(134556767, 999999999);
```

- Valid `ChronoField` values to use with the `get()` method are: `NANO_OF_SECOND`, `MICRO_OF_SECOND`, `MILLI_OF_SECOND`, and `INSTANT_SECONDS`.
- Valid `ChronoUnits` values to use with the `plus()` and `minus()` methods are: `NANOS`, `MICROS`, `MILLIS`, `SECONDS`, `MINUTES`, `HOURS`, `HALF_DAYS`, and `DAYS`.
- `Period` represents an amount of time in terms of years, months and days. You can create an instance using:

```
Period.of(3, 12, 30);
```

- Valid `ChronoUnits` values to use with the `get()` method are: `DAYS`, `MONTHS`, `YEARS`.
- `Duration` represents an amount of time in terms of seconds and nanoseconds. You can create an instance using:

```
Duration.ofSeconds(50, 999999);
```

- Valid `ChronoUnits` values to use with the constructor and the `get()`, `plus()`, and `minus()` methods are: `NANOS`, `MICROS`, `MILLIS`, `SECONDS`, `MINUTES`, `HOURS`, `HALF_DAYS`, and `DAYS`.

Self Test

1. Which of the following are valid ways to create a `LocalDate` object?

A. `LocalDate.of(2014);`
B. `LocalDate.with(2014, 1, 30);`
C. `LocalDate.of(2014, 0, 30);`
D. `LocalDate.now().plusDays(5);`

2. Given:

```
LocalDate.of(2014, 1, 2).atTime(14, 30, 59, 999999)
```

Which of the following is the result of executing the above line?
A. A `LocalDate` of `2014-01-02`
B. A `LocalTime` of `14:30:59:999999`
C. A `LocalDateTime` of `2014-01-02 14:30:59:999999`
D. An exception is thrown

3. Which of the following are valid `ChronoUnit` values for `LocalTime`?
A. `YEAR`
B. `NANOS`
C. `DAY`
D. `HALF_DAYS`

4. Which of the following statements are true?
A. `java.time.Period` implements `java.time.temporal.Temporal`
B. `java.time.Instant` implements `java.time.temporal.Temporal`
C. `LocalDate` and `LocalTime` are thread-safe
D. `LocalDateTime.now()` will return the current time in UTC zone

5. Which of the following is a valid way to get the nanoseconds part of an `Instant` object referenced by `i`?
A. `int nanos = i.getNano();`
B. `long nanos = i.get(ChronoField.NANOS);`
C. `long nanos = i.get(ChronoUnit.NANOS);`
D. `int nanos = i.getEpochNano();`

6. Given:

```
System.out.println(
    Period.between(
        LocalDate.of(2015, 3, 20),
        LocalDate.of(2015, 2, 20))
);
```

Which of the following is the result of executing the above line?
A. `P29D`
B. `P-29D`
C. `P1M`
D. `P-1M`

7. Given:

```
System.out.println(
    Duration.between(
        LocalDateTime.of(2015, 3, 20, 18, 0),
        LocalTime.of(18, 5) )
);
```

Which of the following is the result of executing the above line?
A. `PT5M`
B. `PT-5M`
C. `PT300S`
D. An exception is thrown

8. Which of the following are valid `ChronoField` values for `LocalDate`?
A. `DAY_OF_WEEK`
B. `HOUR_OF_DAY`

C. `DAY_OF_MONTH`
D. `MILLI_OF_SECOND`

Open answers page

Do you like what you read? Would you consider?

Buying the print/kindle version from Amazon

Buying the PDF/EPUB/MOBI versions from Leanpub

Buying the e-book version from iTunes

Buying the e-book version from Kobo

Buying the e-book version from Scribd

Do you have a problem or something to say?

Report an issue with the book

Contact me

---

20. Assertions 22. Time Zones and Daylight Savings